

# Postgres and SQL database

---

- **Postgres and SQL database**
  - **Why Not NoSQL ??**
  - **Why SQL ??**
  - **Creating a database**
  - **Using a library that lets you connect and put data in it**
  - **Creating a table and defining its schema**
  - **Interacting with the database**
    - **CREATE**
    - **UPDATE**
    - **DELETE**
    - **SELECT**
  - **How to do queries from a Node.js app ??**
    - **about client.query**
    - **Code for writing doing CRUD operation from the database**
    - **SQL Injection**
  - **Relationships and Transactions**
    - **Transaction in Postgres SQL**
  - **Joins**
    - **JOINS**
    - **Types of JOINS**
      - **INNER JOIN**
      - **LEFT JOIN**
      - **RIGHT JOIN**
      - **FULL JOIN**

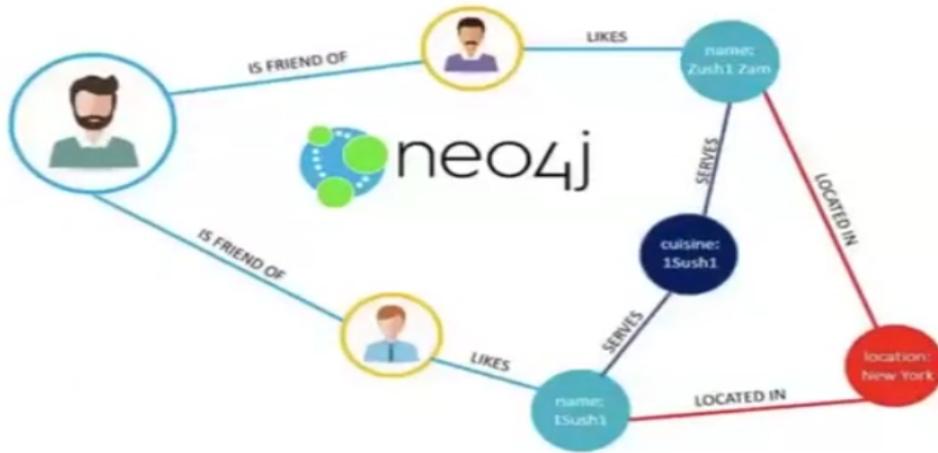
There are different types of databases :-

## 1. NoSQL databases

- Store data in a **schema-less** fashion. Extremely lean and fast way to store data.
- Examples - MongoDB, Firebase, etc..

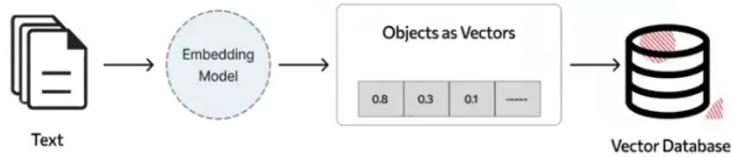
## 2. Graph databases

- Data is stored in the form of a graph. Specially useful in cases **where relationships need to be stored (social networks)**
- Examples - Neo4j



### 3. Vector Databases

- Stores data in the form of vectors
- Useful in Machine learning
- Examples - Pinecone



graft.

### 4. SQL Databases

- **Stores data in the form of rows** (similar to **collection** in **mongoDB**)
- Most full stack applications will use this
- Examples - MySQL, Postgres, Prometheus, etc..

Table: customers

customer_id	first_name	last_name	phone	country
1	John	Doe	817-646-8833	USA
2	Robert	Luna	412-862-0502	USA
3	David	Robinson	208-340-7906	UK
4	John	Reinhardt	307-242-6285	UK
5	Betty	Taylor	806-749-2958	UAE

Why Not NoSQL ??

You might've used MongoDB

It's schemaless properties make it ideal to for bootstrapping a project fast But as your app grows, this property makes it very easy for data to get **corrupted**

### What is Schemaless ??

Different rows can have different **schema** (keys / types)



```
1 _id: ObjectId('65b3f3319e01786d275501c8')
2 userId: 65b3f3319e01786d275501c6
3 balance: 5885.875967139374
4 __v: 0
```



```
_id: ObjectId('65b3f3469e01786d275501ce')
userId: ObjectId('65b3f3469e01786d275501cc')
__v: 0
amountBalance: 720.3759487457523
```



```
_id: ObjectId('65b3f3499e01786d275501d3')
userId: ObjectId('65b3f3499e01786d275501d1')
balance: ""harkirat"""
__v: 0
```

Notice in the above pic, user ne **balance** ki jagah **amountBalance** daal diya then also the **mongoDB** is not complaining, also in the other pic, you can see **amount** me **string** bhej diya(instead of **number**)

### Problems ->

1. Can lead to **inconsistent database**
2. Can cause **runtime errors**
3. Is **too flexible for an app that needs strictness**

### Upsides or Advantages ->

1. Can **move very fast**
2. Can **change schema very easily**



You might think that mongoose does add strictness to the codebase because we used to define a schema there. That strictness is present at the Node.js level, not at the DB level. You can still put in erroneous data in the database that doesn't follow that schema.

## Why SQL ??

SQL databases have a strict schema. They require you to :- (**This is also the STEPs to start using the database**)

1. Define your schema
2. Put in data that follows that schema
3. Update the schema as your app changes and perform **migrations**

So there are 4 parts when using an SQL database (not connecting it to Node.js, just running it and putting data in it)

1. Running the database.

2. Using a library that lets you connect and put data in it.
3. Creating a table and defining its schema.
4. Run queries on the database to interact with the data (Insert/Update/Delete)

## Creating a database

---

You can start a PostgreSQL database in a few ways -

1. **Using neonDB** (easiest way to get `postgres` database) + go to [Neon DB](#) [this is a decent service that lets you create a server]
  - o click on `new project`
  - o give the name to the project
  - o finally click on `create project`
  - o after this you will come to a window where you can find the connection string for PostgreSQL (it is similar to that provided and similar to that in `mongoDB`) [Notice the "\*" present in the string is where you are going to put the password]
2. **Using docker locally**
  - o `docker run --name my-postgres -e POSTGRES_PASSWORD=mysecretpassword -d -p 5432:5432 postgres`
  - o Connection String is below
  - o `postgresql://postgres:mysecretpassword@localhost:5432/postgres?sslmode=disable`
3. **Using docker on windows**
  - o How to run PostgreSQL in windows terminal (if you have docker installed).
    - first run docker GUI application that helps in running commands in terminal.
    - After that run it with the docker instance by the help of following command present after this
    - for the first time if the image is not downloaded use the below command
    - `docker run --name my-postgres1 -e POSTGRES_PASSWORD=mysecretpassword -d -p 5432:5432 postgres`
    - if the docker image is there, prior to use it can simply be runned by `docker run <image name>`
    - After that,
    - use `docker exec -It my-postgres1 psql -U postgres -d postgres` this command in terminal.
    - then enter the password and it will connect to localhost PostgreSQL instance.
    - now you will be inside the PostgreSQL command line that looks like `postgres-#`
    - You can check it by running `\dt`, (the command to display all the tables.)



The connection string is similar to the string we had in mongoose.

Let's understand about the **url or connection string**

---

`postgresql://username:password@host/database`

## Using a library that lets you connect and put data in it

---

1. **psql** **psql** is a terminal-based front-end to PostgreSQL. It provides an interactive command-line interface to the PostgreSQL (or TimescaleDB) database. With psql, you can type in queries interactively, issue them to PostgreSQL, and see the query results.

### How to connect to your database ?

**psql** Comes bundled with postgresql. You don't need it for this tutorial. We will directly be communicating with the database from Node.js

```
psql -h p-br0ken-frost-69135494.us-east-2.aws.neon.tech -d database_name_here -U
IO0xdevs
```

talking about the above url ->

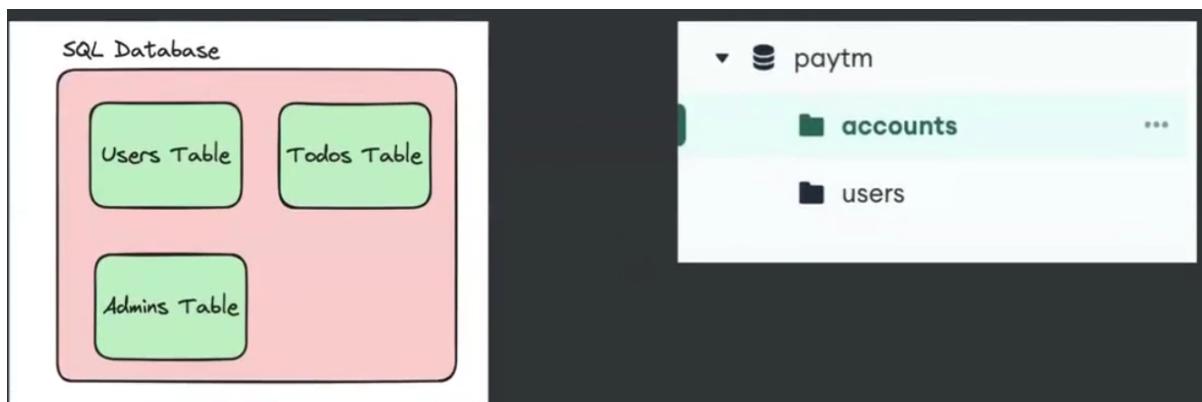
- **-h** -> means where is this hosted ?? you can see this in the connection string generated from **neonDB** after @
- 1. **pg** **pg** is a **Nodejs** library that you can use in your backend app to store data in the Postgres DB (similar to **mongoose** ). We will be installing this eventually in our app.

## Creating a table and defining its schema

---

### Tables in SQL

A single database can have multiple tables inside. Think of them as collections in **MongoDB** database.



 In **mongoDB**, it is known as **COLLECTIONS** and in **MySQL**, it is known as **TABLES**

Until now, we have a database that we can interact with. The next step in case of postgres is to define the **schema** of your tables.

SQL stands for **Structured query language**. It is a language in which you can describe what/how you want to put data in the database.

To create a table, the command to run is -

```
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
```

```
email VARCHAR(255) UNIQUE NOT NULL,  
password VARCHAR(255) NOT NULL,  
created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP  
);
```

There are few parts of this SQL statement, lets decode them one by one

## 1. CREATE TABLE users

`CREATE TABLE users` -> This command initiates the creation of a new table in the database named `users`

## 2. id SERIAL PRIMARY KEY

- `id` -> The **name of the first column in the users table**, typically **used as a unique identifier for each row (user)**. Similar to `_id` in mongodb
- `SERIAL` -> A PostgreSQL-specific data type for **creating an auto-incrementing integer**. *Every time a new row is inserted, this value automatically increments, ensuring each user has a unique id* .
- `PRIMARY KEY` : This constraint specifies that the **id column is the primary key for the table, meaning it uniquely identifies each row**. *Values in this column must be unique and not null.*

## 3. email VARCHAR(255) UNIQUE NOT NULL

- `email` -> The name of the second column, intended to store the user's username.
- `VARCHAR(50)` -> A variable character string data type that can **store up to 50 characters**. It's used here to *limit the length of the username*.
- `UNIQUE` : This constraint ensures that **all values in the username column are unique across the table**. *No two users can have the same username*.
- `NOT NULL` : This constraint **prevents null values from being inserted into the username column**. *Every row must have a username value*.

## 4. password VARCHAR(255) UNIQUE NOT NULL

same as above, can be non unique

## 5. created\_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT\_TIMESTAMP

- `created_at` -> The name of the fifth column, intended to **store the timestamp when the user was created**.
- `TIMESTAMP WITH TIME ZONE` -> This data type **stores both a timestamp and a time zone**, allowing for the precise tracking of when an event occurred, regardless of the user's or server's time zone.
- `DEFAULT CURRENT_TIMESTAMP` -> This default value **automatically sets the created\_at column to the date and time at which the row is inserted into the table, using the current timestamp of the database server**.

If you have made the database then try running the above code for creating the table and updating it

# Interacting with the database

There are 4 things you'd like to do with database

## CREATE

```
INSERT INTO users (username, email, password)
VALUES ('username_here', 'user@example.com', 'user_password');
```



Notice how you didn't have to specify the `id` because it auto increments

## UPDATE

```
UPDATE users
SET password = "new_password"
WHERE email = 'user@example.com';
```

You can add more complexity to the above case

```
UPDATE users
SET password = 'new_password', username= '123123'
WHERE email = 'harkirat@gmail.com' AND username= 'harkirat' ;
```

The above SQL query will first check the user which has `username = 'harkirat'` and `email = 'harkirat@gmail.com'` and then it will update its corresponding password to be '`new_password`' and its `username` to be '`123123`' from '`harkirat`'

## DELETE

```
DELETE FROM users
WHERE id = 1;
```

## SELECT

**used to show the column from the table**

```
SELECT * FROM users
WHERE id = 1;
```

\* will print all the columns present in the table

if you want specific column to be selected and shown then

```
SELECT username, email FROM users;
```

this will show all the `username`, `email` present in the `users` table.

## How to do queries from a Node.js app ??

---

In the end, postgres exposes a protocol that someone needs to talk to be able to send these commands (update, delete) to the database.

`psql` is one such library that takes commands from your terminal and sends it over to the database.

To do the same in a Node.js , you can use one of many `Postgres clients` similar in `mongoDB`. for example -> `mongoose` is one of `mongoDB` client

**pg library**(one of the postgres client)

[pg library](#)

Non - blocking(thread is not blocked for the process like `async`, `await` and others...) postgresSQL client for Node.js

Documentation -> <https://node-postgres.com/>

to use `pg`, after initialising the `node` project with `typescript` enabled write the below command (you know how to do this)

```
npm install pg
```

if using `typescript` then

```
npm install pg @types/pg
```

connecting to the postgres db

```
import { Client } from 'pg'

// as Client is CLASS so to use it first make the instance of the class

const pgClient = new Client("connection_string_to_connect_your_postgres_db") // Either you do this OR
const client = new Client({
  host: 'my.database-server.com', // present in your connection string, anything after '@' is a part of the host
```

```

bort: 5334,
database: 'database-name',
user: 'database-user', // present in your connection string
password: 'secretpassword!!', // present in your connection string
ssl : true // use this only when inside the connection string, sslmode =
require written
})
// As Client can take any of the 2 arguments (one is the connection STRING and
second is providing all the data in OBJECT form and then connect to the database)

pgClient.connect() // REMEMBER this is asynchronous function as this is the code to
connect to the string (SO IT WILL TAKE TIME) // wrap inside the async function

async function main() {
  await pgClient.connect()
}

```

for example :-

```

import {Client} from "pg"

const pgClient = new Client ("postgresql://neondb_owner:wrWG5KI1ziYB@ep-lucky-
snow-a50ilb0b5.us-east-2.aws.neon.tech/neondb?sslmode=require?")

// if above is the connection string then its corresponding OBJECT type will be

const pgClient = new Client({
  user : "neondb_owner",
  password : "wrWG5KI1ziYB",
  port : 5432,
  host : "ep-lucky-snow-a50ilb0b5.us-east-2.aws.neon.tech",
  database : "neondb"
  ssl : true // as our connection string consists of sslmode = require
})
// all of the value has been or can be extracted from the connection string only
except port value

```

## Now QUERYING part

### about client.query

-> whatever the command or sql query you used to give it to any sql database, that query can **be inserted into this function inside " " or "" and then this will execute the command written inside it** [do all the CRUD operations like this]

```

const result = await client.query('SELECT * FROM USERS;')
console.log(result)

```

## Write a function to create a users table in your database

```
import {Client} from "pg"

const pgClient = new Client({
  connectionString : "postgresql://your connection string goes here"
})

async function createUsersTable() { // as yahan par server se len den ki baat ho rhi h so use "await" as much as possible
  await pgClient.connect()
  const result = await pgClient.query(`CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
  );
`)
  console.log(result) // Will give the table as well as whole other things
  console.log(result.rows) // Will give all the entry present in the table
}

createUsersTable();
```

## Code for writing doing CRUD operation from the database

Now how do you actually use it in your website to store the important things in the database. A demo of it is given below:-

```
import express from "express"
import { Client} from "pg";

const app = express();
app.use(express.json());

const pgClient = new Client("postgresql://neondb_owner:wrWG5KI1ziYB@ep-lucky-snow-a50ilb0b5.us-east-2.aws.neon.tech/neondb?sslmode=require?")

pgClient.connect();

app.post("/signup", async (req, res) => {
  const username = req.body.username
  const password = req.body.password
  const email = req.body.email

  let sqlQuery = "INSERT INTO users (username, email, password) VALUES (" + username + ", " + email + ", " + password + ")"
  sqlQuery += " ON CONFLICT DO UPDATE SET password = EXCLUDED.password";
```

```

        sqlQuery += ",";
        sqlQuery += email;
        sqlQuery += ",";
        sqlQuery += password;
        sqlQuery += ");"
    
```

// EITHER WRITE LIKE THE ABOVE or WRITE LIKE THE BELOW (just in one line)  
[BOTH will work]

```

const insertQuery = `INSERT INTO users (username, email, password) VALUES
('${username}', '${email}', '${password}')` // 2

const response = await pgClient.query(insertQuery)

res.json({
    message : "You have signed up"
})
}

app.listen(3000)

```

**⚠ Remember ->** always put all the variables values inside the `' '` or `" "` (see the `// 2` line of code) while writing the SQL Query

Ouput of the above code ->

	Id	username	email	password	created_at
12	12	harkirat	user@example.com	user_password	2024-11-30 15:18:46.69279+00
13	13	asdasd	asdasd	asdasd	2024-11-30 15:57:00.954487+00
15	15	harkirat123123	123123	123123	2024-11-30 16:07:04.737837+00

see in the right pic, you got the message "You have signed up" and if you see to the neon db site then under the `users` table you can see the new entry data of what has been sent from the `postman` (see `15` entry of the table)

**💡 What are the problems here in the above code ?**

**1. Backend Crashes ->** If you send something which is against the rules defined when creating the table (like username should be unique etc..) the your **Backend crashes**

- to fix it -> **put it inside the TRY-CATCH Block**

```

import express from "express"
import { Client } from "pg";

```

```

const app = express();
app.use(express.json());

const pgClient = new Client("postgresql://neondb_owner:wrWG5KI1ziYB@ep-lucky-snow-a50ilb0b5.us-east-2.aws.neon.tech/neondb?sslmode=require")

async function main(){
    await pgClient.connect();
}

main()

app.post("/signup", async (req, res) => {
    const username = req.body.username
    const password = req.body.password
    const email = req.body.email

    try {
        const insertQuery = `INSERT INTO users (username, email, password) VALUES ('${username}', '${email}', '${password}')`;

        const response = await pgClient.query(insertQuery)

        res.json({
            message : "You have signed up"
        })
    }catch(e){
        res.status(400).json({
            message : "Error while signing up"
        })
    }
}

app.listen(3000)

```

putting it inside the **TRY-CATCH** block will solve the things but you will see that this will still give the ERROR Reason is the 2nd thing to keep in mind

## 2. SQL Injection

### SQL Injection

---

A term you have widely heard when it comes to hacking any website or steal some data from it.

Now you will be sending all the things like **username**, **password** and **email** inside the browser and eventually it will go to some endpoint. Now this endpoint can be extracted from the developer's mode and then through **postman**, you can send the request with all the credentials required (**username**, **password** and **email**). Here **SQL Injection** comes into the picture.

User can send **ABNOXIOUS thing (or bad data)** from the **postman**

something like this :-



```
1 {  
2     "username": {  
3         "aa": "asdads",  
4         "asdasd": "dadsdas"  
5     }  
6 }
```

so you can say that **I WILL USE ZOD to validate the INPUT** and their **types** (with some constraint)

BUT **ZOD** will help you in the case of **NoSQL** database, here it cannot help much as something like **SQL injection** exists here.

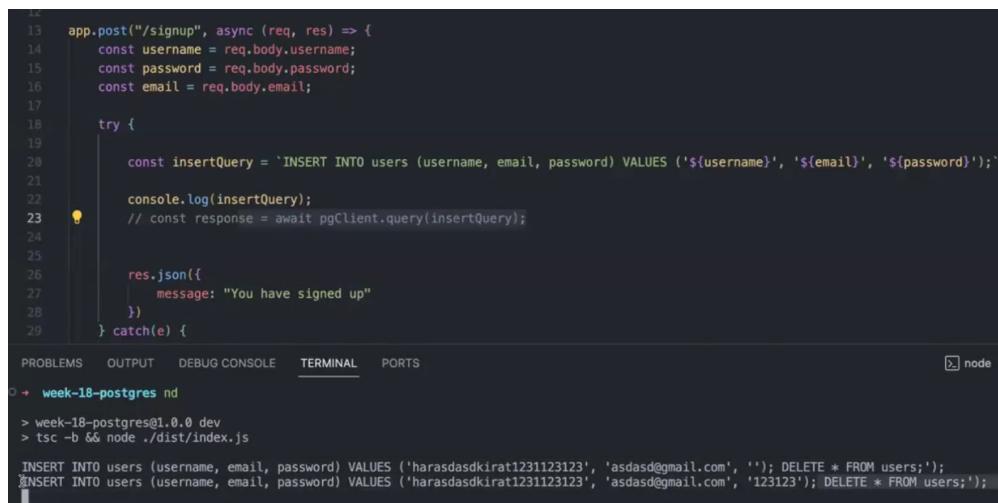
Now user has the capability to set **any password** so he can **inject SQL inside the password and do something which will make your website vulnerable**



```
"username": "harasdasdkirat1231123123",  
"email": "asdasd@gmail.com",  
"password": "123123'); DELETE * FROM users;"
```

User has injected a query to delete all things from the database and with the help of password

if you try to console what is actually being queried to your server and eventually to your backend it looks something like this



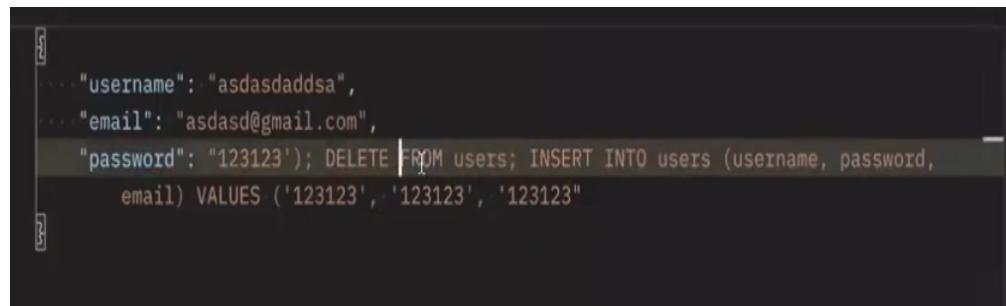
```
13 app.post("/signup", async (req, res) => {  
14     const username = req.body.username;  
15     const password = req.body.password;  
16     const email = req.body.email;  
17  
18     try {  
19  
20         const insertQuery = `INSERT INTO users (username, email, password) VALUES ('${username}', '${email}', '${password}')`;  
21  
22         console.log(insertQuery);  
23         // const response = await pgClient.query(insertQuery);  
24  
25  
26         res.json({  
27             message: "You have signed up"  
28         })  
29     } catch(e) {  
30     }  
31 }  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100  
101  
102  
103  
104  
105  
106  
107  
108  
109  
110  
111  
112  
113  
114  
115  
116  
117  
118  
119  
120  
121  
122  
123  
124  
125  
126  
127  
128  
129  
130  
131  
132  
133  
134  
135  
136  
137  
138  
139  
140  
141  
142  
143  
144  
145  
146  
147  
148  
149  
150  
151  
152  
153  
154  
155  
156  
157  
158  
159  
160  
161  
162  
163  
164  
165  
166  
167  
168  
169  
170  
171  
172  
173  
174  
175  
176  
177  
178  
179  
180  
181  
182  
183  
184  
185  
186  
187  
188  
189  
190  
191  
192  
193  
194  
195  
196  
197  
198  
199  
200  
201  
202  
203  
204  
205  
206  
207  
208  
209  
210  
211  
212  
213  
214  
215  
216  
217  
218  
219  
220  
221  
222  
223  
224  
225  
226  
227  
228  
229  
230  
231  
232  
233  
234  
235  
236  
237  
238  
239  
240  
241  
242  
243  
244  
245  
246  
247  
248  
249  
250  
251  
252  
253  
254  
255  
256  
257  
258  
259  
260  
261  
262  
263  
264  
265  
266  
267  
268  
269  
270  
271  
272  
273  
274  
275  
276  
277  
278  
279  
280  
281  
282  
283  
284  
285  
286  
287  
288  
289  
290  
291  
292  
293  
294  
295  
296  
297  
298  
299  
300  
301  
302  
303  
304  
305  
306  
307  
308  
309  
310  
311  
312  
313  
314  
315  
316  
317  
318  
319  
320  
321  
322  
323  
324  
325  
326  
327  
328  
329  
330  
331  
332  
333  
334  
335  
336  
337  
338  
339  
340  
341  
342  
343  
344  
345  
346  
347  
348  
349  
350  
351  
352  
353  
354  
355  
356  
357  
358  
359  
360  
361  
362  
363  
364  
365  
366  
367  
368  
369  
370  
371  
372  
373  
374  
375  
376  
377  
378  
379  
380  
381  
382  
383  
384  
385  
386  
387  
388  
389  
390  
391  
392  
393  
394  
395  
396  
397  
398  
399  
400  
401  
402  
403  
404  
405  
406  
407  
408  
409  
410  
411  
412  
413  
414  
415  
416  
417  
418  
419  
420  
421  
422  
423  
424  
425  
426  
427  
428  
429  
430  
431  
432  
433  
434  
435  
436  
437  
438  
439  
440  
441  
442  
443  
444  
445  
446  
447  
448  
449  
450  
451  
452  
453  
454  
455  
456  
457  
458  
459  
460  
461  
462  
463  
464  
465  
466  
467  
468  
469  
470  
471  
472  
473  
474  
475  
476  
477  
478  
479  
480  
481  
482  
483  
484  
485  
486  
487  
488  
489  
490  
491  
492  
493  
494  
495  
496  
497  
498  
499  
500  
501  
502  
503  
504  
505  
506  
507  
508  
509  
510  
511  
512  
513  
514  
515  
516  
517  
518  
519  
520  
521  
522  
523  
524  
525  
526  
527  
528  
529  
530  
531  
532  
533  
534  
535  
536  
537  
538  
539  
540  
541  
542  
543  
544  
545  
546  
547  
548  
549  
550  
551  
552  
553  
554  
555  
556  
557  
558  
559  
5510  
5511  
5512  
5513  
5514  
5515  
5516  
5517  
5518  
5519  
5520  
5521  
5522  
5523  
5524  
5525  
5526  
5527  
5528  
5529  
5530  
5531  
5532  
5533  
5534  
5535  
5536  
5537  
5538  
5539  
55310  
55311  
55312  
55313  
55314  
55315  
55316  
55317  
55318  
55319  
55320  
55321  
55322  
55323  
55324  
55325  
55326  
55327  
55328  
55329  
55330  
55331  
55332  
55333  
55334  
55335  
55336  
55337  
55338  
55339  
55340  
55341  
55342  
55343  
55344  
55345  
55346  
55347  
55348  
55349  
55350  
55351  
55352  
55353  
55354  
55355  
55356  
55357  
55358  
55359  
55360  
55361  
55362  
55363  
55364  
55365  
55366  
55367  
55368  
55369  
55370  
55371  
55372  
55373  
55374  
55375  
55376  
55377  
55378  
55379  
55380  
55381  
55382  
55383  
55384  
55385  
55386  
55387  
55388  
55389  
55390  
55391  
55392  
55393  
55394  
55395  
55396  
55397  
55398  
55399  
553100  
553101  
553102  
553103  
553104  
553105  
553106  
553107  
553108  
553109  
553110  
553111  
553112  
553113  
553114  
553115  
553116  
553117  
553118  
553119  
553120  
553121  
553122  
553123  
553124  
553125  
553126  
553127  
553128  
553129  
553130  
553131  
553132  
553133  
553134  
553135  
553136  
553137  
553138  
553139  
553140  
553141  
553142  
553143  
553144  
553145  
553146  
553147  
553148  
553149  
553150  
553151  
553152  
553153  
553154  
553155  
553156  
553157  
553158  
553159  
553160  
553161  
553162  
553163  
553164  
553165  
553166  
553167  
553168  
553169  
553170  
553171  
553172  
553173  
553174  
553175  
553176  
553177  
553178  
553179  
553180  
553181  
553182  
553183  
553184  
553185  
553186  
553187  
553188  
553189  
553190  
553191  
553192  
553193  
553194  
553195  
553196  
553197  
553198  
553199  
553200  
553201  
553202  
553203  
553204  
553205  
553206  
553207  
553208  
553209  
553210  
553211  
553212  
553213  
553214  
553215  
553216  
553217  
553218  
553219  
553220  
553221  
553222  
553223  
553224  
553225  
553226  
553227  
553228  
553229  
553230  
553231  
553232  
553233  
553234  
553235  
553236  
553237  
553238  
553239  
553240  
553241  
553242  
553243  
553244  
553245  
553246  
553247  
553248  
553249  
553250  
553251  
553252  
553253  
553254  
553255  
553256  
553257  
553258  
553259  
553260  
553261  
553262  
553263  
553264  
553265  
553266  
553267  
553268  
553269  
553270  
553271  
553272  
553273  
553274  
553275  
553276  
553277  
553278  
553279  
553280  
553281  
553282  
553283  
553284  
553285  
553286  
553287  
553288  
553289  
553290  
553291  
553292  
553293  
553294  
553295  
553296  
553297  
553298  
553299  
5532100  
5532101  
5532102  
5532103  
5532104  
5532105  
5532106  
5532107  
5532108  
5532109  
5532110  
5532111  
5532112  
5532113  
5532114  
5532115  
5532116  
5532117  
5532118  
5532119  
55321100  
55321101  
55321102  
55321103  
55321104  
55321105  
55321106  
55321107  
55321108  
55321109  
55321110  
55321111  
55321112  
55321113  
55321114  
55321115  
55321116  
55321117  
55321118  
55321119  
553211100  
553211101  
553211102  
553211103  
553211104  
553211105  
553211106  
553211107  
553211108  
553211109  
553211110  
553211111  
553211112  
553211113  
553211114  
553211115  
553211116  
553211117  
553211118  
553211119  
5532111100  
5532111101  
5532111102  
5532111103  
5532111104  
5532111105  
5532111106  
5532111107  
5532111108  
5532111109  
5532111110  
5532111111  
5532111112  
5532111113  
5532111114  
5532111115  
5532111116  
5532111117  
5532111118  
5532111119  
55321111100  
55321111101  
55321111102  
55321111103  
55321111104  
55321111105  
55321111106  
55321111107  
55321111108  
55321111109  
55321111110  
55321111111  
55321111112  
55321111113  
55321111114  
55321111115  
55321111116  
55321111117  
55321111118  
55321111119  
553211111100  
553211111101  
553211111102  
553211111103  
553211111104  
553211111105  
553211111106  
553211111107  
553211111108  
553211111109  
553211111110  
553211111111  
553211111112  
553211111113  
553211111114  
553211111115  
553211111116  
553211111117  
553211111118  
553211111119  
5532111111100  
5532111111101  
5532111111102  
5532111111103  
5532111111104  
5532111111105  
5532111111106  
5532111111107  
5532111111108  
5532111111109  
5532111111110  
5532111111111  
5532111111112  
5532111111113  
5532111111114  
5532111111115  
5532111111116  
5532111111117  
5532111111118  
5532111111119  
55321111111100  
55321111111101  
55321111111102  
55321111111103  
55321111111104  
55321111111105  
55321111111106  
55321111111107  
55321111111108  
55321111111109  
55321111111110  
55321111111111  
55321111111112  
55321111111113  
55321111111114  
55321111111115  
55321111111116  
55321111111117  
55321111111118  
55321111111119  
553211111111100  
553211111111101  
553211111111102  
553211111111103  
553211111111104  
553211111111105  
553211111111106  
553211111111107  
553211111111108  
553211111111109  
553211111111110  
553211111111111  
553211111111112  
553211111111113  
553211111111114  
553211111111115  
553211111111116  
553211111111117  
553211111111118  
553211111111119  
5532111111111100  
5532111111111101  
5532111111111102  
5532111111111103  
5532111111111104  
5532111111111105  
5532111111111106  
5532111111111107  
5532111111111108  
5532111111111109  
5532111111111110  
5532111111111111  
5532111111111112  
5532111111111113  
5532111111111114  
5532111111111115  
5532111111111116  
5532111111111117  
5532111111111118  
5532111111111119  
55321111111111100  
55321111111111101  
55321111111111102  
55321111111111103  
55321111111111104  
55321111111111105  
55321111111111106  
55321111111111107  
55321111111111108  
55321111111111109  
55321111111111110  
55321111111111111  
55321111111111112  
55321111111111113  
55321111111111114  
55321111111111115  
55321111111111116  
55321111111111117  
55321111111111118  
55321111111111119  
553211111111111100  
553211111111111101  
553211111111111102  
553211111111111103  
553211111111111104  
553211111111111105  
553211111111111106  
553211111111111107  
553211111111111108  
553211111111111109  
553211111111111110  
553211111111111111  
553211111111111112  
553211111111111113  
553211111111111114  
553211111111111115  
553211111111111116  
553211111111111117  
553211111111111118  
553211111111111119  
5532111111111111100  
5532111111111111101  
5532111111111111102  
5532111111111111103  
5532111111111111104  
5532111111111111105  
5532111111111111106  
5532111111111111107  
5532111111111111108  
5532111111111111109  
5532111111111111110  
5532111111111111111  
5532111111111111112  
5532111111111111113  
5532111111111111114  
5532111111111111115  
5532111111111111116  
5532111111111111117  
5532111111111111118  
5532111111111111119  
55321111111111111100  
55321111111111111101  
55321111111111111102  
55321111111111111103  
55321111111111111104  
55321111111111111105  
55321111111111111106  
55321111111111111107  
55321111111111111108  
55321111111111111109  
55321111111111111110  
55321111111111111111  
55321111111111111112  
55321111111111111113  
55321111111111111114  
55321111111111111115  
55321111111111111116  
55321111111111111117  
55321111111111111118  
55321111111111111119  
553211111111111111100  
553211111111111111101  
553211111111111111102  
553211111111111111103  
553211111111111111104  
553211111111111111105  
553211111111111111106  
553211111111111111107  
553211111111111111108  
553211111111111111109  
553211111111111111110  
553211111111111111111  
553211111111111111112  
553211111111111111113  
553211111111111111114  
553211111111111111115  
553211111111111111116  
553211111111111111117  
553211111111111111118  
553211111111111111119  
5532111111111111111100  
5532111111111111111101  
5532111111111111111102  
5532111111111111111103  
5532111111111111111104  
5532111111111111111105  
5532111111111111111106  
5532111111111111111107  
5532111111111111111108  
5532111111111111111109  
5532111111111111111110  
5532111111111111111111  
5532111111111111111112  
5532111111111111111113  
5532111111111111111114  
5532111111111111111115  
5532111111111111111116  
5532111111111111111117  
5532111111111111111118  
5532111111111111111119  
5
```

Ignore the first query it was for some other password, look at the second query which prints on console. Now go to the sql editor and paste this query you will see that

1. users table me corresponding email, username, and password insert ho jayega and then
2. users table se sare records DELETE ho jayega as you have written the second query in that way only

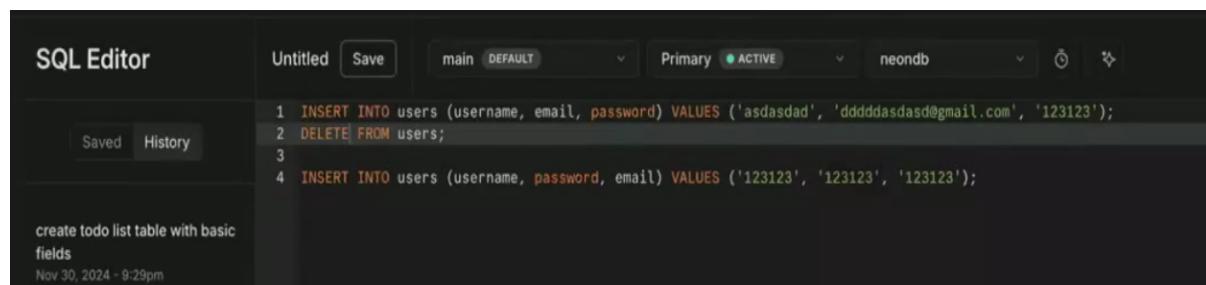
### Can you see how much vulnerable our app or website has became ??

Another example (More better way sirf query he h isme nothing extra characters which will lead to run the query without even getting the error)



```
... "username": "asdasdaddsa",
... "email": "asdasd@gmail.com",
"password": "123123'); DELETE FROM users; INSERT INTO users (username, password,
email) VALUES ('123123', '123123', '123123"
```

Now you can see its totally valid (3 queries h isme) will look something like this



SQL Editor

```
Untitled Save main DEFAULT Primary ACTIVE neondb
1 INSERT INTO users (username, email, password) VALUES ('asdasdad', 'dddddasd@gmail.com', '123123');
2 DELETE FROM users;
3
4 INSERT INTO users (username, password, email) VALUES ('123123', '123123', '123123');
```

create todo list table with basic fields  
Nov 30, 2024 - 9:29pm

Now the above statement will execute like this manner ->

- first it will insert the given data in the users table
- second **it will delete all the things inside the users table**
- third it will insert the data in the users table(and only this data now exists in the users table)

With this power, it can do anything with the database (**CRUD operations basically**) and without even getting notices (**Think how dangerous it is**)

### 💡 How to fix this ??

-> **Pretty easy actually, you can achieve it by just changing the way you write the query**

```
import express from "express"
import { Client } from "pg";

const app = express();
app.use(express.json());

const pgClient = new Client("postgresql://neondb_owner:wrWG5KI1ziYB@ep-lucky-snow-
```

```
a50i1b0b5.us-east-2.aws.neon.tech/neondb?sslmode=require?")

async function main(){
    await pgClient.connect();
}

main()

app.post("/signup", async (req, res) => {
    const username = req.body.username
    const password = req.body.password
    const email = req.body.email

    try {
        // Instead of writing like this
        // const insertQuery = `INSERT INTO users (username, email, password)
VALUES ('${username}', '${email}', '${password}')`;
        // WRITE LIKE THIS
        const insertQuery = `INSERT INTO users (username, email, password) VALUES
(\$1, \$2, \$3);` 

        // and then adding the value of 1, 2 and 3 inside the response variable
        // like this (GIVE IT INSIDE THE ARRAY)
        const values = [username, email, password]
        const response = await pgClient.query(insertQuery, values)
        // Benefit of writing like this see // 2

        res.json({
            message : "You have signed up"
        })
    }catch(e){
        res.status(400).json({
            message : "Error while signing up"
        })
    }
})

app.listen(3000)
```

**Explanation of // 2 code** By writing like this you have separated the way data was going inside the sql

INSERT INTO users (username, email, password) VALUES (\$1, \$2, \$3) -> ye alg se pahunchega database me and value of username, email, password alg se pahunchega. now isse hogya kya ki values of all these 3 will be **STORED NOT APPENDED inside the original string due to which IT GET NEVER RUN**

will look something like this in the database

	id	username	email	password	created_at
	29	123123	123123	123123	2024-11-30 16:16:20.8228
	30	asdasdaddsa	asdasd@gmail..	123123'); DELETE FROM users; INSERT INTO users (user..	2024-11-30 16:19:14.1064

Notice the sql query **got stored not executed and hence you prevented the SQL injection**

In this way, you can also find that someone was trying to inject the **sql** inside the database

### Access token and Refresh token

-> **Problem with JWTs** -> If your jwt gets leaked then anyone can access your account, there is no way to revoke JWT. so to solve this, you can store the JST in the database and check if it has revoked

BUT it has problem too -> you are hitting the database too much and with every request you are checking the database that whether the JWT has been revoked or not

so **ACCESS and REFRESH token works as Middleground** ->

- **REFRESH token are stored inside the database while ACCESS token aren't** so the user send the request with Access token and Refresh token. If their access token is valid for now(usually it is valid for 1 hour) then you let them through and not hit the database but if the access token is not valid then you hit the database, fetch the database, use the refresh token and then again generate a new access token

### Why this is great ?

-> If the access token gets leaked then they only have that for an hour (after an hour, it will become invalid) and if the refresh token gets leaked then they can revoke it as they are stored inside the database.

## Relationships and Transactions

---

if you know this then you can try to build [app100x](#) [ becomes tough as it has **Nested structure**]

if it would have been using **mongoDb** as the database then the schema would have looked similar to the below image

```
{
  courseId: 1,
  title: "Full Stack course"
  folders: [
    {
      name: "Week 1",
      children: [
        {
          type: "folder",
          name: "Week 1 pre-requisites",
          children: [
            {name: "HTML", videoUrl: "api.100xdevs.com/video/1mp4"}
          ]
        }
      ]
    }
  ]
}
```

although for a **nested structure type site**, its better to use the **NoSQL** database as you dont know the limit of nesting (it can be infinitely nested i.e. you can add more nesting here)

This is similar to what we have studied in **mongoDb** relationships

**Relationships let you store data in different tables and relate it with each other.**

Although you can put all the things inside one table in the **NoSQL** database (which is the advantage as well as disadvantage for it) lets understand it by taking the example of **to-do**

```
users
[
  {
    name: "Harkirat",
    email: "harkirat@gmail.com",
    todos: []
  }
]
```

you can see you have inserted all the todos inside an array and that is indirectly inside a single table known as `users` table.

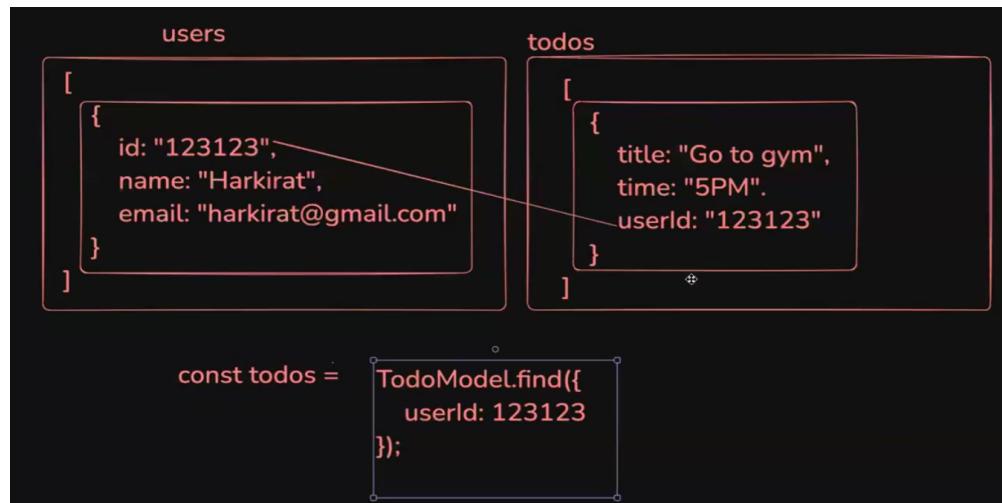
### Upside of this ->

- the schema is easier to write and read
- sb kuch ek jagah pr h

### Downside of this ->

- any user first has to go inside the `users` table to access `todos` which is not good

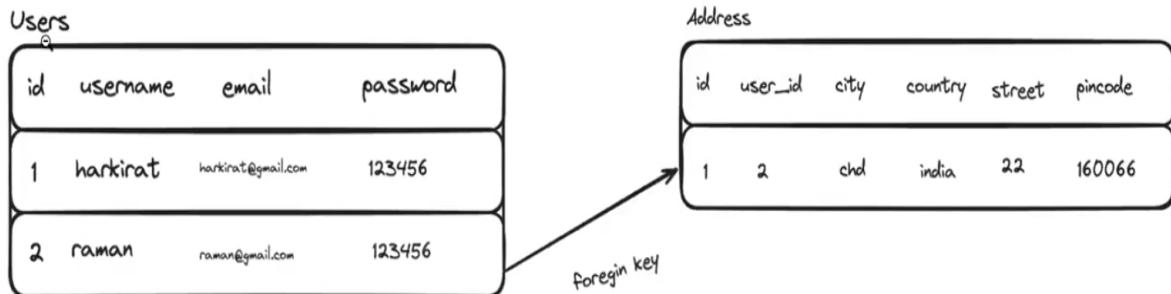
hence **Decomposing data(seperating them) is good thing** so you should seperate both `todos` and `users` table and then **try to relate them** something like the below



Notice the **Relationship denoted by RED line (userId related to id)** and hence we were able to find out the user `todos` with the help of the id and **relationship which has been created**

**If you are seperating tables, then you must ensure that the RELATIONSHIPS should exist among the tables created**

so **relationships in SQL database looks like this ->**



This is called a **relationship**, which means that the **Address** table is related to the **Users** table.

When defining the table, you need to define the **relationships**

```

CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    email VARCHAR(255) UNIQUE NOT NULL,
    password VARCHAR(255) NOT NULL,
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE addresses(
    id SERIAL PRIMARY KEY,
    user_id INTEGER NOT NULL,
    city VARCHAR(100) NOT NULL,
    country VARCHAR(100) NOT NULL,
    street VARCHAR(255) NOT NULL,
    pincode VARCHAR(20),
    created_at TIMESTAMP WITH TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(id) ON DELETE CASCADE // at last you
have to write the relationship
    // This means that addresses table ka jo user_id field h that REFERENCES to
the id of the users table
    // ON DELETE CASCADE -> agar kbhi users table me kuch delete hua to jahan
jahan uska relationship exist krta h wahan wahan pr(here addresses table) me jake
v delete kr do
    // similar to this is
    // ON DELETE RESTRICT -> this will restrict you to delete FIRST all the
relationship which exists for the main table and then only you can delete an entry
in the main table
);

```

## SQL Query

To insert the address of a user ->

```

INSERT INTO addresses (user_id, city, country, street, pincode)
VALUES (1, 'New York', 'USA', '123 Broadway St', '10001');

```

Now if you want to get the address of a user given an **id**, you can run the following query

```
SELECT city, country, street, pincode
FROM addresses
WHERE user_id = 1;
```

Lets try to define all the above things inside and call it from the **node.js** app

```
import express from "express"
import { Client } from "pg";

const app = express();
app.use(express.json());

const pgClient = new Client("postgresql://neondb_owner:wrWG5KI1ziYB@ep-lucky-snow-a50ilb0b5.us-east-2.aws.neon.tech/neondb?sslmode=require?")

async function main(){
    await pgClient.connect();
}

main()

app.post("/signup", async (req, res) => {
    const username = req.body.username
    const password = req.body.password
    const email = req.body.email

    const city = req.body.city
    const country = req.body.country
    const street = req.body.street
    const pincode = req.body.pincode

    try {
        // Making the users table and then inserting into it
        const insertQuery = `INSERT INTO users (username, email, password) VALUES ($1, $2, $3) RETURNING id`;
        // RETURNING id means -> id of the new user which has just now been inserted will be returned by writing this line of code
        // id which got returned from the above line, store it inside the userId
        const userId = response.rows[0].id

        const values = [username, email, password]
        const response = await pgClient.query(insertQuery, values) // 2

        // Making the address table and then inserting into it
        const addressInsertQuery = `INSERT INTO addresses (city, country, street,
```

```

pincode, user_id) VALUES ($1, $2, $3, $4, $5);` 

    const addressInsertResponse = await pgClient.query(addressInsertQuery,
[city, country, street, pincode, userId])

    res.json({
        message : "You have signed up"
    })
}catch(e){
    res.status(400).json({
        message : "Error while signing up"
    })
}
}

app.listen(3000)

```

### The above code has some issues, can you see this ??

-> see the // 2 line of code, lets suppose there is case that after execution of // 2 code, server crashed or lets say database me time lg gya iske baad(next wale line ko execute krne me), in this case, `users` table to ban gya data v chla gya but `addresses` table to hold pe chla gya na.

Now for a website, this is one **of the critical issue you will often face**

A more practical scenario of this case will be `Paytm`

```

app.post("/signup", async(req, res) => {
    const query1 = "UPDATE USER WHERE user_id = 1 SET balance = balance - 10" // 1
    const query2 = "UPDATE USER WHERE user_id = 2 SET balance = balance + 10" // 2
    ek se kat ke dusre me jayega
})

```

Now here if the above case will come then ek query to excute ho gya but dusra nhi hua (**means // 1 execute ho gya but // 2 nhi**), this will totally create the chaos among the users.

This is where `Transaction` comes into the picture.

## Transaction in Postgres SQL

**Whenever you have 2 or more queries which are interrelated to each other very strictly (means ek ke change hone se dusre me change hona JARURI he h), in this case YOU SHOULD WRAP the queries inside the TRANSACTION**

### Transactions in SQL

 Good question to have at this point is what queries are run when the user signs up and sends both their information and their address in a single request.

**Do we send two SQL queries into the database? What if one of the queries (address query for example) fails? This would require **transactions** in SQL to ensure either both the user information and address goes in, or neither does**

```
BEGIN; <-- Start transaction

INSERT INTO users (username, email, password)
VALUES ('john_doe','john_doe@example.com','securepassword123');

INSERT INTO addresses (user_id, city, country, street, pincode)
VALUES (currval('users_id_seq'),'New York', 'USA', '123 Broadway St', '10001');

COMMIT; <-- End transaction
```

you have basically wrapped both the **Insert** code inside the TRANSACTION

Applying the above concept to above **paytm** example ->

```
app.post("/signup", async(req, res) => {
    const query3 = "BEGIN;"
    const query1 = "UPDATE USER WHERE user_id = 1 SET balance = balance - 10" // 1
    const query2 = "UPDATE USER WHERE user_id = 2 SET balance = balance + 10" // 2
    const query4 = "COMMIT;"
})
```

just by wrapping inside the **BEGIN** and **COMMIT** you have said that **all the queries which are present inside this block, EITHER THEY ALL WILL EXECUTE OR NONE OF THEM WILL EXECUTE (at a particular point of time). Remember A(Atomicity) in ACID property of database**

simply saying, if **// 1** has executed and **// 2** failed, then **// 1** will **Revert back**

wrapping the above code(example of **node.js** site) inside the transaction

```
import express from "express"
import { Client } from "pg";

const app = express();
app.use(express.json());

const pgClient = new Client("postgresql://neondb_owner:wrWG5KI1ziYB@ep-lucky-snow-a50i1b0b5.us-east-2.aws.neon.tech/neondb?sslmode=require")

async function main(){
    await pgClient.connect();
}

main()
```

```

app.post("/signup", async (req, res) => {
  const username = req.body.username
  const password = req.body.password
  const email = req.body.email

  const city = req.body.city
  const country = req.body.country
  const street = req.body.street
  const pincode = req.body.pincode

  try {

    const insertQuery = `INSERT INTO users (username, email, password) VALUES
    ($1, $2, $3) RETURNING id;`
    const addressInsertQuery = `INSERT INTO addresses (city, country, street,
    pincode, user_id) VALUES ($1, $2, $3, $4, $5);` // 1

    // wrapped inside the transaction
    await pgClient.query("BEGIN;")
    const values = [username, email, password]
    const response = await pgClient.query(insertQuery, values)
    const userId = response.rows[0].id // 2
    const addressInsertResponse = await pgClient.query(addressInsertQuery,
    [city, country, street, pincode, userId])
    await pgClient.query("COMMIT;")

    res.json({
      message : "You have signed up"
    })
  }catch(e){
    res.status(400).json({
      message : "Error while signing up"
    })
  }
}

app.listen(3000)

```

 You can have question in your mind that if the server crashed at // 1 line of code, then // 2 will be able to get id as RETURNING id to execute ho gya but // 2 to chla he nhi so in this case it tries to create the new entry and eventually fails

## Joins

---

Lets try to solve a problem



💡 given the id = 12, return me all the data(means user related info. + address related info.) related to user whose id = 12 if someone hits the backend point (/metadata) ??

-> how will you do the above problem

```

import express from "express"
import { Client } from "pg";

const app = express();
app.use(express.json());

const pgClient = new Client("postgresql://neondb_owner:wrWG5KI1ziYB@ep-lucky-snow-a50ilb0b5.us-east-2.aws.neon.tech/neondb?sslmode=require?")

async function main(){
    await pgClient.connect();
}

main()

app.post("/signup", async (req, res) => {
    // signup logic already written above
})

app.get("/metadata", async (req, res) => {
    const id = req.body.id // given this id = 12 here how to get all the data

    // const query1 = await `SELECT * FROM users WHERE id = $1` // make sure you
    // do not add "*" as it will return all the data (including some sensitive data such
    // as password) so instead jitna jarurat h utna he return kro
    const query1 = await `SELECT username, email, id FROM users WHERE id = $1`
    const response1 = pgClient.query(query1, [id]);

    const query2 = await `SELECT * FROM addresses WHERE user_id = $1`
    const response2 = pgClient.query(query2, [id]);

    res.json({
        user : response1.rows[0]
        address : response2.rows[0] // 2
        address : response2.rows // 3
    })
})

```

```
    })
})
app.listen(3000)
```

when you will go to link -> <https://localhost:3000/metadata?id=12>"

you will see the output

```
{
  "user": {
    "username": "1211212312331223123231",
    "email": "adddak12312312312ldkl1k123@gmail.com",
    "id": 17
  },
  "address": {
    "id": 9,
    "user_id": 17,
    "city": "Chandigarh",
    "country": "India",
    "street": "123123",
    "pincode": "12331",
    "created_at": "2024-12-01T15:33:43.964Z"
  }
}
```

You can see all the data related to user with id = 12 in the above picture

But the above code has some error, **ideally user has more than one address, so it should be able to see all the address stored and linked to it in the database and hence you should RETURN AN ARRAY of addresses**

so instead of returning only one row, it should return **all the rows** so write // 3 line of code instead of // 2. (if a new entry will come that will also now will be reflected now)

```
{
  "user": {
    "username": "1211212312331223123231",
    "email": "adddak12312312312ldkl1k123@gmail.com",
    "id": 17
  },
  "address": [
    {
      "id": 9,
      "user_id": 17,
      "city": "Chandigarh",
      "country": "India",
      "street": "123123",
      "pincode": "12331",
      "created_at": "2024-12-01T15:33:43.964Z"
    }
  ]
}
```

Now the output looks like this -> you have got the **ARRAY of address being returned**

 **why this approach is bad ??**

-> due to the reason that you are **sending two separate queries**[if some change happened after the 1st query so 2nd query will not be knowing about this (later it will know but for the time being, it will not be able to know about this), and hence anomalies will occur]

**The Better version of the above approach is JOINS**

## JOINS

---

Defining relationships is easy.

What's hard is Joining data from two (or more) tables together.

**For example,** if I ask you to fetch me a users details and their address, what SQL would you run?

### Approach 1 -> Bad one

- Query 1: Fetch user's details

```
SELECT id, username, email  
FROM users  
WHERE id = YOUR_USER_ID
```

- Query 2 Fetch user's address

```
SELECT city, country, street, pincode  
FROM addresses  
WHERE user_id = YOUR_USER_ID
```

### Approach 2 -> using joins

```
SELECT users.id, users.username, users.email, addresses.city, addresses.country,  
addresses.street, addresses.pincode  
FROM users  
JOIN addresses ON users.id = addresses.user_id  
WHERE usersid = `1`;
```

```
// OR you can write the above code like this also(just used ALIAS in this nothing  
extra)  
SELECT u.id, u.username, u.email, a.city, a.country, a.street, a.pincode  
FROM users u  
JOIN addresses a ON u.id = a.user_id  
WHERE u.id = YOUR_USER_ID
```



**Remember -> while using JOIN, a new table is created.**

users				addresses				
id	name	email	password	id	user_id	city	country	pincode
1.	harkirat	harkirat@gm.com	123123	1.	1.	chd.	ind.	155500
				2.	1.	delhi	ind.	155001

- JOIN of the above table looks like this

users.id	users.name	users.email	users.password	addresses.city	addresses.pincode
1.	harkirat	harkirat@gmail.	123123	chandigarh	155000
1.	harkirat	harkirat@gmail.	123123	delhi	155001

so applying the above concept in solving or making the endpoint -> [/metadata](#)

the code looks like this ->

```
import express from "express"
import { Client } from "pg";

const app = express();
app.use(express.json());

const pgClient = new Client("postgresql://neondb_owner:wrW5KI1ziYB@ep-lucky-snow-a50ilb0b5.us-east-2.aws.neon.tech/neondb?sslmode=require?")

async function main(){
    await pgClient.connect();
}

main()

app.post("/signup", async (req, res) => {
    // signup logic already written above
})

app.get("/metadata", async(req, res) => {
    const id = req.query.id
    const query = `SELECT users.id, users.username, users.email, addresses.city, addresses.country, addresses.street, addresses.pincode FROM users JOIN addresses ON users.id = addresses.user_id WHERE usersid = $1;`
    // Instead of $1 why not this -> ${id} REASON -> SQL Injection
})
```

```

const response = await pgClient.query(query, [id])
res.json({
  response : response.rows // REMEMBER -> if you will do simply send
  "response" then PURA OBJECT de dega (consisting of hell lot of things (from SQL
  command which involved to other SQL realted entity value))
  // by writing "response.rows" -> ye bas table me jo pda h wo dega (NOTHING
EXTRA)
})
})
app.listen(3000)

```

Now the output looks like this ->

```
{
  "response": [
    {
      "id": 17,
      "username": "1211212312331223123231",
      "email": "addak12312312312ldkl1k123@gmail.com",
      "city": "Chandigarh",
      "country": "India",
      "street": "123123",
      "pincode": "12331"
    }
  ]
}
```

 **Remember -> JOIN is a expensive task to perform** [computationally tough task to perform]

lets say one table has M (lets say 10000) rows and other has N(lets say 35) rows then to join both the table **in WORST CASE would be M\*N (10000 \* 35)**

Other than the above demerit, it(JOIN) has mostly advantages :-

1. **Reduced Latency**
2. **Simplified Application Logic**
3. **Transactional Integrity**

## Types of JOINS

---

### INNER JOIN

**Returns rows when there is AT LEAST one match in both tables.** If there is no match, the rows are not returned. It's the **most common type** of join.

**Use Case :-** Find All Users With Their Addresses. If a user hasn't filled their address, that user shouldn't be returned.

 **Remember -> BY DEFAULT -> JOIN works the same way as INNER JOIN**

```
SELECT users.username, addresses.city, addresses.country,  
addresses.street, addresses.pincode  
FROM users  
INNER JOIN addresses ON users.id = addresses.user_id;
```

## LEFT JOIN

**Returns ALL rows from the LEFT table, and the MATCHED rows from the RIGHT table.**

**Use case :-** To **list all users from your database along with their address information** (if they've provided it), you'd use a LEFT JOIN. Users without an address will still appear in your query result, but the address fields will be NULL for them.

```
SELECT users.username, addresses.city, addresses.country, addresses.street,  
addresses.pincode  
FROM users  
LEFT JOIN addresses ON users.id = addresses.user_id;
```

## RIGHT JOIN

**Returns ALL rows from the RIGHT table, and the MATCHED rows from the LEFT table.**

**simply saying -> if there is something which is in left and not in right, then that should come as well as vice versa of this should also come and those which are present in both left and right that should also come**

 **RIGHT JOIN = !(LEFT JOIN)**

**Use case :-** Given the structure of the database, a **RIGHT JOIN would be less common since the addresses table is unlikely to have entries not linked to a user due to the foreign key constraint**. However, if you had a situation where you start with the **addresses** table and optionally include user information, this would be the theoretical use case.

```
SELECT users.username, addresses.city, addresses.country, addresses.street  
addresses.pincode  
FROM users  
RIGHT JOIN addresses ON users.id = addresses.user_id;
```

## FULL JOIN

**Returns rows when there is a MATCH in one of the tables. It effectively combines the results of both LEFT JOIN and RIGHT JOIN.**

 **FULL JOIN = RIGHT JOIN + LEFT JOIN**

**Use case :-** A FULL JOIN would **combine all records from both users and addresses showing the relationship where it exists.** Given the constraints, this might not be as relevant because every **address** should be linked to a **user**, but if there were somehow orphaned records on either side, this query would reveal them.

```
SELECT users.username, addresses.city, addresses.country, addresses.street,  
addresses.pincode  
FROM users  
FULL JOIN addresses ON users.id = addresses.user_id;
```